加微信：642945106 发送“ 赠送 ”领取赠送精品课程
发数字“ 2 ”获取众筹列表

06 | RPC实战：剖析gRPC源码，动手实现一个完整的RPC

2020-03-02 何小锋

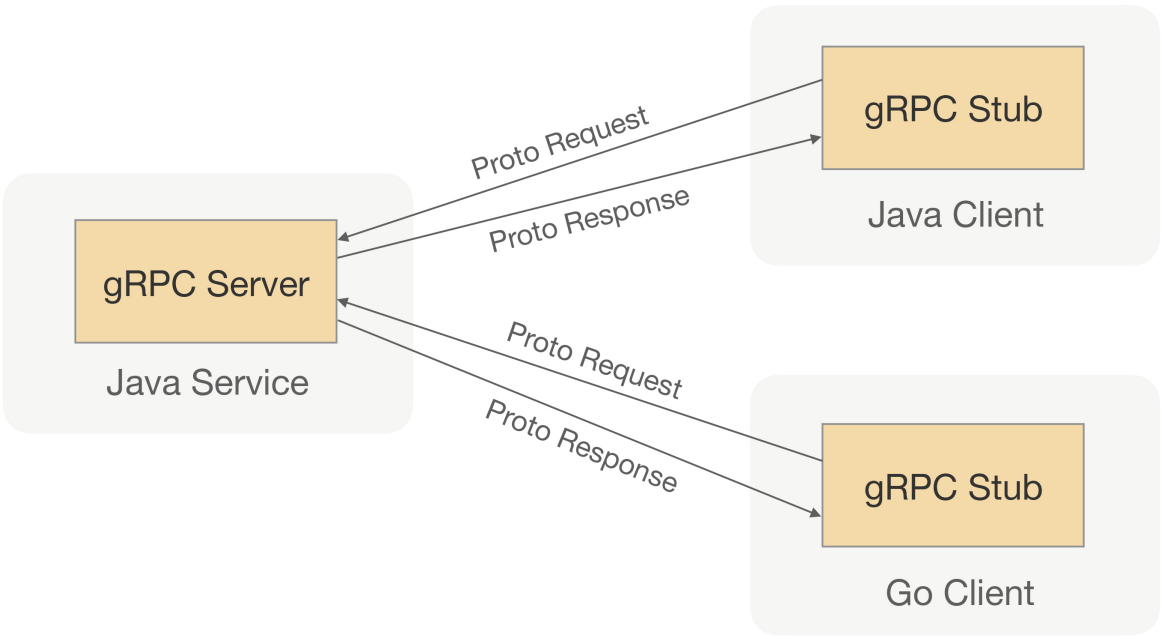
讲述：张浩
时长 11:14　大小 10.30M

你好，我是何小锋。上一讲我分享了动态代理，其作用总结起来就是一句话：“我们可以通过动态代理技术，屏蔽 RPC 调用的细节，从而让使用者能够面向接口编程。”

到今天为止，我们已经把 RPC 通信过程中要用到的所有基础知识都讲了一遍，但这些内容多属于理论。**这一讲我们就来实战一下，看看具体落实到代码上，我们应该怎么实现一个 RPC 框架？**

为了能让咱们快速达成共识，我选择剖析 gRPC 源码（源码地址：https://github.com/grpc/grpc-java）。通过分析 gRPC 的通信过程，我们可以清楚地知道在 gRPC 里面这些知识点是怎么落地到具体代码上的。

gRPC 是由 Google 开发并且开源的一款高性能、跨语言的 RPC 框架，当前支持 C、Java 和 Go 等语言，当前 Java 版本最新 Release 版为 1.27.0。gRPC 有很多特点，比如跨语言，通信协议是基于标准的 HTTP/2 设计的，序列化支持 PB（Protocol Buffer）和 JSON，整个调用示例如下图所示：



gRPC调用示例图

如果你想快速地了解一个全新框架的工作原理，我个人认为最快的方式就是从使用示例开始，所以现在我们就以最简单的 HelloWord 为例开始了解。

在这个例子里面，我们会定义一个 say 方法，调用方通过 gRPC 调用服务提供方，然后服务提供方会返回一个字符串给调用方。

为了保证调用方和服务提供方能够正常通信，我们需要先约定一个通信过程中的契约，也就是我们在 Java 里面说的定义一个接口，这个接口里面只会包含一个 say 方法。在 gRPC 里面定义接口是通过写 Protocol Buffer 代码，从而把接口的定义信息通过 Protocol Buffer 语义表达出来。HelloWord 的 Protocol Buffer 代码如下所示：

复制代码

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "io.grpc.hello";
option java_outer_classname = "HelloProto";
```

```
option objc_class_prefix = "HLW";

package hello;

service HelloService{
rpc Say(HelloRequest) returns (HelloReply) {}
}

message HelloRequest {
string name = 1;
}

message HelloReply {
string message = 1;
}
```

有了这段代码，我们就可以为客户端和服务器端生成消息对象和 RPC 基础代码。我们可以利用 Protocol Buffer 的编译器 protoc，再配合 gRPC Java 插件（protoc-gen-grpc-java），通过命令行 protoc3 加上 plugin 和 proto 目录地址参数，我们就可以生成消息对象和 gRPC 通信所需要的基础代码。如果你的项目是 Maven 工程的话，你还可以直接选择使用 Maven 插件来生成同样的代码。

防止断更 请关注加
首发微信：17161436665

发送原理

生成完基础代码以后，我们就可以基于生成的代码写下调用端代码，具体如下：

复制代码

```
package io.grpc.hello;

import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;
import io.grpc.StatusRuntimeException;


import java.util.concurrent.TimeUnit;

public class HelloWorldClient {

    private final ManagedChannel channel;
    private final HelloServiceGrpc.HelloServiceBlockingStub blockingStub;
    /**
    * 构建Channel连接
    **/
    public HelloWorldClient(String host, int port) {
        this(ManagedChannelBuilder.forAddress(host, port)
```

```
                .usePlaintext()
                .build());
    }

    /**
    * 构建Stub用于发请求
    **/
    HelloWorldClient(ManagedChannel channel) {
        this.channel = channel;
        blockingStub = HelloServiceGrpc.newBlockingStub(channel);
    }

    /**
    * 调用完手动关闭
    **/
    public void shutdown() throws InterruptedException {
        channel.shutdown().awaitTermination(5, TimeUnit.SECONDS);
    }


    /**
    * 发送rpc请求
    **/
    public void say(String name) {
        // 构建入参对象
        HelloRequest request = HelloRequest.newBuilder().setName(name).build()
        HelloReply response;
        try {
            // 发送请求
            response = blockingStub.say(request);
        } catch (StatusRuntimeException e) {
            return;
        }
        System.out.println(response);
    }

    public static void main(String[] args) throws Exception {
            HelloWorldClient client = new HelloWorldClient("127.0.0.1", 50051)
            try {
                client.say("world");
            } finally {
                client.shutdown();
            }
    }
}
```

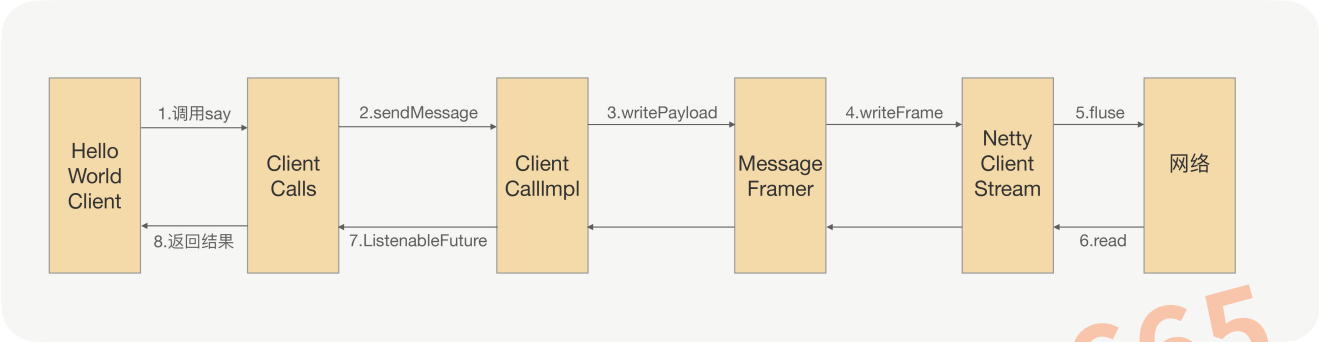
调用端代码大致分成三个步骤：

首先用 host 和 port 生成 channel 连接；

然后用前面生成的 HelloService gRPC 创建 Stub 类；

最后我们可以用生成的这个 Stub 调用 say 方法发起真正的 RPC 调用，后续其它的 RPC 通信细节就对我们使用者透明了。

为了能看清楚里面具体发生了什么，我们需要进入到 ClientCalls.blockingUnaryCall 方法里面看下逻辑细节。但是为了避免太多的细节影响你理解整体流程，我在下面这张图中只画下了最重要的部分。



整体流程图

拼课微信：1716143665

我们可以看到，在调用端代码里面，我们只需要一行（第 48 行）代码就可以发起一个 RPC 调用，而具体这个请求是怎么发送到服务提供者那端的呢？这对于我们 gRPC 使用者来说是完全透明的，我们只要关注是怎么创建出 stub 对象的就可以了。

比如入参是一个字符对象，gRPC 是怎么把这个对象传输到服务提供方的呢？因为在[第 03 讲] 中我们说过，只有二进制才能在网络中传输，但是目前调用端代码的入参是一个字符对象，那在 gRPC 里面我们是怎么把对象转成二进制数据的呢？

回到上面流程图的第 3 步，在 writePayload 之前，ClientCallImpl 里面有一行代码就是 method.streamRequest(message)，看方法签名我们大概就知道它是用来把对象转成一个 InputStream，有了 InputStream 我们就很容易获得入参对象的二进制数据。这个方法返回值很有意思，就是为啥不直接返回我们想要的二进制数组，而是返回一个 InputStream 对象呢？你可以先停下来想下原因，我们会在最后继续讨论这个问题。

我们接着看 streamRequest 方法的拥有者 method 是个什么对象？我们可以看到 method 是 MethodDescriptor 对象关联的一个实例，而 MethodDescriptor 是用来存放要调用 RPC 服务的接口名、方法名、服务调用的方式以及请求和响应的序列化和反序列化实现类。

大白话说就是，MethodDescriptor 是用来存储一些 RPC 调用过程中的元数据，而在 MethodDescriptor 里面 requestMarshaller 是在绑定请求的时候用来序列化方式对象的，所以当我们调用 method.streamRequest(message) 的时候，实际是调用 requestMarshaller.stream(requestMessage) 方法，而 requestMarshaller 里面会绑定一个 Parser，这个 Parser 才真正地把对象转成了 InputStream 对象。

讲完序列化在 gRPC 里面的应用后，我们再来看下在 gRPC 里面是怎么完成请求数据“断句”的，就是我们在[第 02 讲] 中说的那个问题——二进制流经过网络传输后，怎么正确地还原请求前语义？

我们在 gRPC 文档中可以看到，gRPC 的通信协议是基于标准的 HTTP/2 设计的，而 HTTP/2 相对于常用的 HTTP/1.X 来说，它最大的特点就是多路复用、双向流，该怎么理解这个特点呢？这就好比我们生活中的单行道和双行道，HTTP/1.X 就是单行道，HTTP/2 就是双行道。

那既然在请求收到后需要进行请求“断句”，那肯定就需要在发送的时候把断句的符号加上，我们看下在 gRPC 里面是怎么加的？

因为 gRPC 是基于 HTTP/2 协议，而 HTTP/2 传输基本单位是 Frame，Frame 格式是以固定 9 字节长度的 header，后面加上不定长的 payload 组成，协议格式如下图所示：

Bit Offset	0-71	72-95	96-103	104-111	112-112
0	header	长度	类型	标示	保留位
帧ID	113-143				
帧内容	payload				

那在 gRPC 里面就变成怎么构造一个 HTTP/2 的 Frame 了。

现在回看我们上面那个流程图的第 4 步，在 write 到 Netty 里面之前，我们看到在 MessageFramer.writePayload 方法里面会间接调用 writeKnownLengthUncompressed 方法，该方法要做的两件事情就是构造 Frame Header 和 Frame Body，然后再把构造的

Frame 发送到 NettyClientHandler，最后将 Frame 写入到 HTTP/2 Stream 中，完成请求消息的发送。

接收原理

讲完 gRPC 的请求发送原理，我们再来看下服务提供方收到请求后会怎么处理？我们还是接着前面的那个例子，先看下服务提供方代码，具体如下：

复制代码

```
static class HelloServiceImpl extends HelloServiceGrpc.HelloServiceImplBase {

  @Override
  public void say(HelloRequest req, StreamObserver<HelloReply> responseObserve
    HelloReply reply = HelloReply.newBuilder().setMessage("Hello " + req.getNa
    responseObserver.onNext(reply);
    responseObserver.onCompleted();
  }
}
```

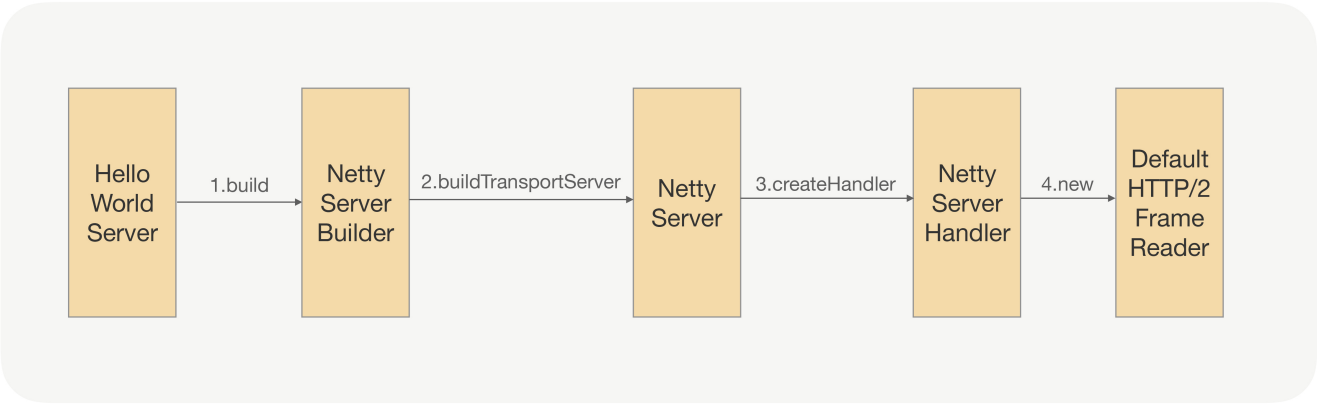
上面 HelloServiceImpl 类是按照 gRPC 使用方式实现了 HelloService 接口逻辑，但是对于调用者来说并不能把它调用过来，因为我们没有把这个接口对外暴露，在 gRPC 里面我们是采用 Build 模式对底层服务进行绑定，具体代码如下：

复制代码

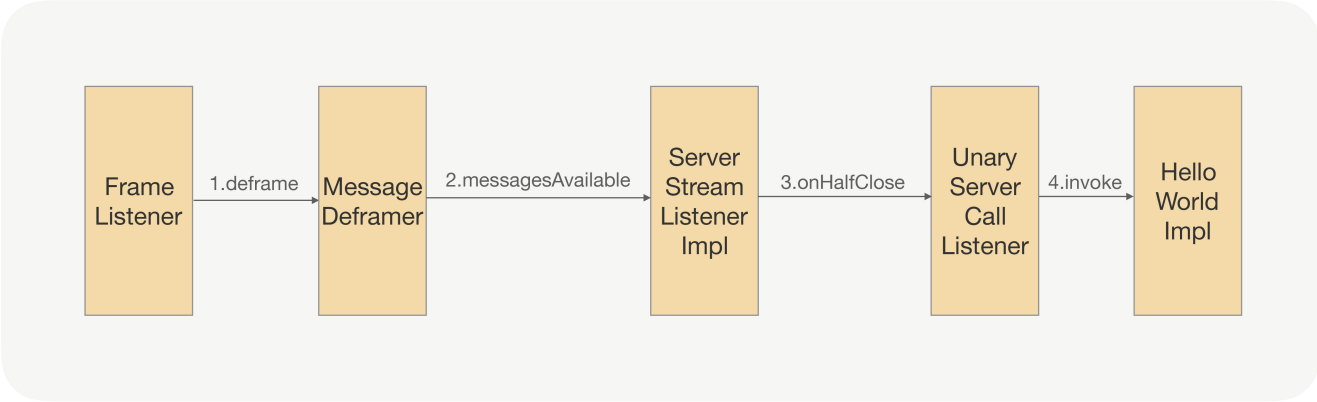
```
package io.grpc.hello;

import io.grpc.Server;
import io.grpc.ServerBuilder;
import io.grpc.stub.StreamObserver;

import java.io.IOException;


public class HelloWorldServer {

  private Server server;

  /**
  * 对外暴露服务
  **/
  private void start() throws IOException {
    int port = 50051;
    server = ServerBuilder.forPort(port)
        .addService(new HelloServiceImpl())
```

```
21          .build()
22          .start();
23      Runtime.getRuntime().addShutdownHook(new Thread() {
24        @Override
25        public void run() {
26          HelloWorldServer.this.stop();
27        }
28      });
29    }
30
31    /**
32    * 关闭端口
33    **/
34    private void stop() {
35      if (server != null) {
36        server.shutdown();
37      }
38    }
39
40    /**
41    * 优雅关闭
42    **/
43    private void blockUntilShutdown() throws InterruptedException {
44      if (server != null) {
45        server.awaitTermination();
46      }
47    }
48
49
50    public static void main(String[] args) throws IOException, InterruptedExcept
51      final HelloWorldServer server = new HelloWorldServer();
52      server.start();
53      server.blockUntilShutdown();
54    }
55
56  }
```

服务对外暴露的目的是让过来的请求在被还原成信息后，能找到对应接口的实现。在这之前，我们需要先保证能正常接收请求，通俗地讲就是要先开启一个 TCP 端口，让调用方可以建立连接，并把二进制数据发送到这个连接通道里面，这里依然只展示最重要的部分。



这四个步骤是用来开启一个 Netty Server，并绑定编解码逻辑的，如果你暂时看不懂，没关系的，我们可以先忽略细节。我们重点看下 NettyServerHandler 就行了，在这个 Handler 里面会绑定一个 FrameListener，gRPC 会在这个 Listener 里面处理收到数据请求的 Header 和 Body，并且也会处理 Ping、RST 命令等，具体流程如下图所示：



在收到 Header 或者 Body 二进制数据后，NettyServerHandler 上绑定的 FrameListener 会把这些二进制数据转到 MessageDeframer 里面，从而实现 gRPC 协议消息的解析 。

那你可能会问，这些 Header 和 Body 数据是怎么分离出来的呢？按照我们前面说的，调用方发过来的是一串二进制数据，这就是我们前面开启 Netty Server 的时候绑定 Default HTTP/2FrameReader 的作用，它能帮助我们按照 HTTP/2 协议的格式自动切分出 Header 和 Body 数据来，而对我们上层应用 gRPC 来说，它可以直接拿拆分后的数据来用。

总结

这是我们基础篇的最后一讲，我们采用剖析 gRPC 源码的方式来学习如何实现一个完整的 RPC。当然整个 gRPC 的代码量可比这多得多，但今天的主要目就是想让你把前面所学的

序列化、协议等方面的知识落实到具体代码上，所以我们这儿只分析了 gRPC 收发请求两个过程。

实现了这两个过程，我们就可以完成一个点对点的 RPC 功能，但在实际使用的时候，我们的服务提供方通常都是以一个集群的方式对外提供服务的，所以在 gRPC 里面你还可以看到负载均衡、服务发现等功能。而且 gRPC 采用的是 HTTP/2 协议，我们还可以通过 Stream 方式来调用服务，以提升调用性能。

总的来说，其实我们可以简单地认为 **gRPC 就是采用 HTTP/2 协议，并且默认采用 PB 序列化方式的一种 RPC**，它充分利用了 HTTP/2 的多路复用特性，使得我们可以在同一条链路上双向发送不同的 Stream 数据，以解决 HTTP/1.X 存在的性能问题。

课后思考

我们讲到，在 gRPC 调用的时候，我们有一个关键步骤就是把对象转成可传输的二进制，但是在 gRPC 里面，我们并没有直接转成二进制数组，而是返回一个 InputStream，你知道这样做的好处是什么吗？

欢迎留言和我分享你的答案，也欢迎你把文章分享给你的朋友，邀请他加入学习。我们下节课再见！

更多课程推荐

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇　05 | 动态代理：面向接口编程，屏蔽RPC处理流程

精选留言 (3)

写留言

新世界
2020-03-02

老师能否把代码提交到github上，我们可以下载下来跑一跑，看一看

cricket1981
2020-03-02

以流的方式处理请求数据，适合请求数据数据量大情况。好比Sax和Dom区别。

高源
2020-03-02

我猜的是网络字节序问题吧😊

展开